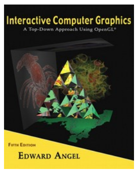

Interactive Computer Graphics
A Top-Down Approach Using OpenGL
Fifth Edition
Edward Angel

Name: ______________________ Term # ____

Homework 6 SOLUTIONS

(400 points)

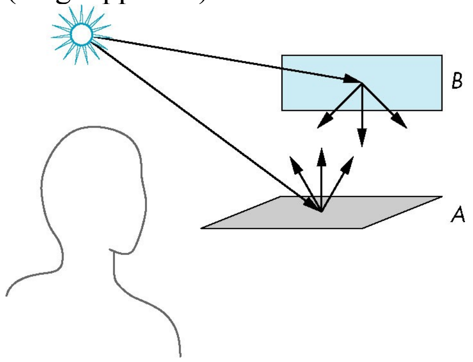
NOTE: **Chapter 6 of the textbook shows the point lighting and surface shading modeling.**
Part A is intended to be done by hand.
Part B is an OpenGL application.

A. **(300 pts) Paper and Pencil**

***(Guidelines:** Read the material from the textbook chapter, you can use textbook figures to exemplify your answer, use keywords, summarize your answer, but the answer **cannot be longer the 7 lines!**)*

6.1 LIGHT AND MATTER

a. Explain Figure 6.1 on Reflecting Surfaces.
ANSWER:
(Angel pp. 290)



6.1 LIGHT AND MATTER In Chapters 1 and 2, we presented the rudiments of human color vision, delaying until now any discussion of the interaction between light and surfaces. Perhaps the most general approach to rendering is based on physics, where we use principles such as conservation of energy to derive equations that describe how light interacts with materials. From a physical perspective, a surface can either emit light by self- emission, as a light bulb does, or reflect light from other surfaces that illuminate it. Some surfaces may both reflect light and emit light from internal physical processes. When we look at a point on an object, the color that we see is determined by multiple interactions among light sources and reflective surfaces. These interactions can be viewed as a re-cursive process. Consider the simple scene illustrated in Figure 6.1. Some light from the source that reaches surface A is scattered. Some of this reflected light reaches sur-face B, and some of it is then scattered back to A, where some of it is again reflected back to B, and so on. This recursive scattering of light between surfaces accounts for subtle shading effects, such as the bleeding of colors between adjacent surfaces. Mathematically, this recursive process results in an integral equation, the rendering equation, which in principle we could use to find the shading of all surfaces in a scene. Unfortunately, this equation generally cannot be solved analytically. Numerical methods are not fast enough for real- time rendering. There are various approximate approaches, such as radiosity and ray tracing, each of which is an excellent approx-imation to the rendering equation for particular types of surfaces. Unfortunately, neither ray tracing nor radiosity can yet be used to render scenes at the rate at which we can pass polygons through the modeling- projection pipeline. Consequently, we Because we only need to consider light that enters the camera by passing through the center of projection, we can start at the center of projection and follow a projec-tor though each pixel in the clipping window. If we assume that all our surfaces are opaque, then the color of the first surface intersected along each projector determines the color of the corresponding pixel in the frame buffer. Note that most rays leaving a light source do not contribute to the image and are thus of no interest to us. Hence, by starting at the center of projection and casting rays we have the start of an efficient ray tracing method, one that we explore further in Section 6.10 and Chapter 13.

b. Explain Figure 6.2 on Light and Surfaces.
ANSWER:
(Angel pp. 291)

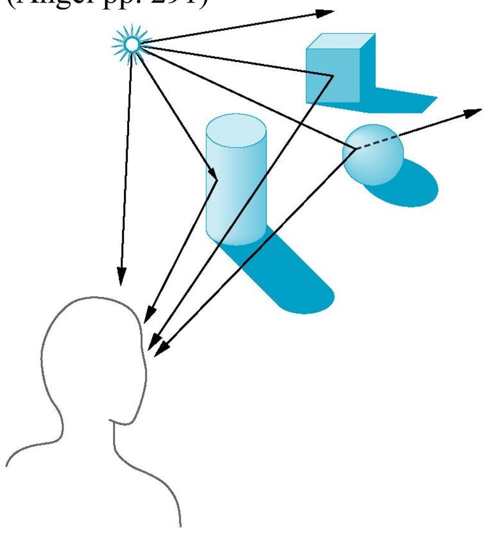
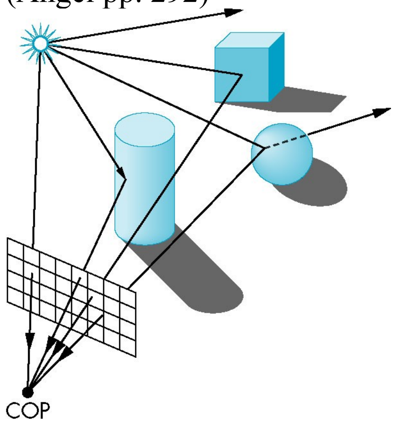


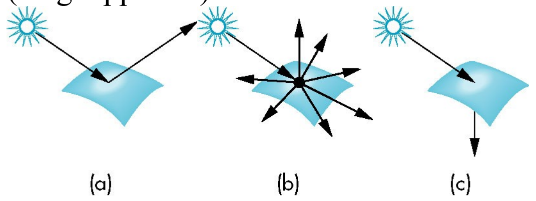
Figure 6.2 shows both single and multiple interactions between rays and objects. It is the nature of these interactions that determines whether an object appears red or brown, light or dark, dull or shiny. When light strikes a surface, some of it is absorbed and some of it is reflected. If the surface is opaque, reflection and absorption account for all the light striking the surface. If the surface is translucent, some of the light is transmitted through the material and emerges to interact with other objects. These interactions depend on the wavelength of the light. An object illuminated by white light appears red because it absorbs most of the incident light but reflects light in the red range of frequencies. A shiny object appears so because its surface is smooth. Conversely, a dull object has a rough surface. The shading of objects also depends on the orientation of their surfaces, a factor that we will see is characterized by the normal vector at each point.

c. Explain Figure 6.3 on Light,Surfaces, and computer imaging.
ANSWER:
(Angel pp. 292)



These interactions between light and materials can be classified into the three groups depicted in Figure 6.4.

d. Explain Figure 6.4 on Light-material interaction.
ANSWER:
(Angel pp. 293)



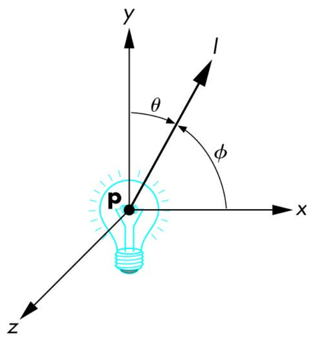
1. Specular surfaces appear shiny because most of the light that is reflected or scattered is in a narrow range of angles close to the angle of reflection. Mirrors are perfectly specular surfaces; the light from an incoming light ray may be partially absorbed, but all reflected light emerges at a single angle, obeying the rule that the angle of incidence is equal to the angle of reflection.

2. Diffuse surfaces are characterized by reflected light being scattered in all directions. Walls painted with matte or flat paint are diffuse reflectors, as are many natural materials, such as terrain viewed from an airplane or a satellite. Perfectly diffuse surfaces scatter light equally in all directions and thus appear the same to all viewers.

3. Translucent surfaces allow some light to penetrate the surface and to emerge from another location on the object. This process of refraction characterizes glass and water. Some incident light may also be reflected at the surface.
From a physical perspective, the reflection, absorption, and transmission of light at the surface of a material is described by a single function called the Bidirectional Reflection Distribution Function (BRDF). Consider a point on a surface. Light energy potentially can arrive at this location from any direction and be reflected and leave in any direction. Thus, for every pair of input and output directions, there will be a value that is the fraction of the incoming light that is reflected in the output direction. The BDRF is a function of five variables: the frequency of light, the two angles required to describe the direction of the input vector, and the two angles required to describe the direction of the output vector. For a perfectly diffuse surface, the BDRF is simplified because it will have the same value for all possible output vectors. For a perfect reflector, the BRDF will only be nonzero when the angle of incidence is equal to the angle of reflection. A real physical surface is never perfectly diffuse nor perfectly specular. Although we could approach light material interactions by modeling the BDRF for materials of interest, the subject is not only complex but does not lead to efficient computational models. Instead, we use models that lead to fast computations, especially when we employ them in a pipeline architecture. These models are based on adding together the contributions of simple models of diffuse, specular, and ambient reflections and, if necessary, accounting for transmission. We develop models for these idealized interactions in Section 6.3 and Chapter 9.

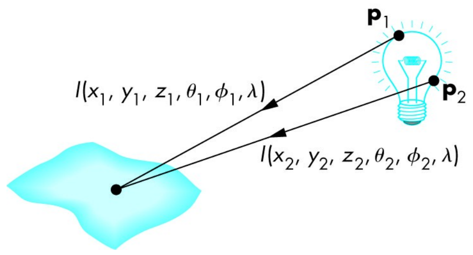
6.2 LIGHT SOURCES

a. Explain Figure 6.5 on Light Sources.
ANSWER:
(Angel pp. 294)



6.2 LIGHT SOURCES Light can leave a surface through two fundamental processes: self- emission and reflection. We usually think of a light source as an object that emits light only through internal energy sources. However, a light source, such as a light bulb, can also reflect some light that is incident on it from the surrounding environment. We neglect the emissive term in our simple models. When we discuss OpenGL lighting in Section 6.7, we will see that we can add a self- emission term. If we consider a source such as the one shown in Figure 6.5, we can look at it as an object with a surface. Each point (x, y, z) on the surface can emit light that is characterized by the direction of emission (. , f) and the intensity of energy emitted at each wavelength .. Thus, a general light source can be characterized by a six- variable illumination function I(x, y, z , . , f, .). Note that we need two angles to specify a direction, and we are assuming that each frequency can be considered independently.

b. Explain Figure 6.6 on Adding the Contribution to a Source.
ANSWER:
(Angel pp. 294)


$\mathbf{p}_1$
$I(x_1, y_1, z_1, \theta_1, \phi_1, \lambda)$
$\mathbf{p}_2$
$I(x_2, y_2, z_2, \theta_2, \phi_2, \lambda)$

From the perspective of a surface illuminated by this source, we can obtain the total contribution of the source (Figure 6.6) by integrating over its surface, a process that accounts for the emission angles that reach this surface and must also account for the distance between the source and the surface. For a distributed light source, such as a light bulb, the evaluation of this integral is difficult, whether we use analytic or numerical methods. Often, it is easier to model the distributed source with polygons, each of which is a simple source, or with an approximating set of point sources.

6.2.1 Color Sources

a. Explain the concept of color sources.
ANSWER:
(Angel pp. 295)
Not only do light sources emit different amounts of light at different frequencies, but their directional properties can vary with frequency. Consequently, a physically correct model can be complex. However, our model of the human visual system is based on three- color theory that tells us we perceive three tristimulus values, rather than a full color distribution. For most applications, we can thus model light sources as having three components red, green, and blue and can use each of the three color sources to obtain the corresponding color component that a human observer sees. Thus, we can describe a color source through a three- component intensity or illumination function L = (Lr , Lg , Lb), each of whose components is the intensity of the independent red, green, and blue components. Thus, we use the red component of a light source for the calculation of the red component of the image. Because light material computations involve three similar but independent calculations, we tend to present a single scalar equation, with the understanding that it can represent any of the three color components. Rather than write what will turn out to be identical expressions for each component of L, we will use the the scalar L to denote any of the its components. That is, L . { Lr , Lg , Lb}.

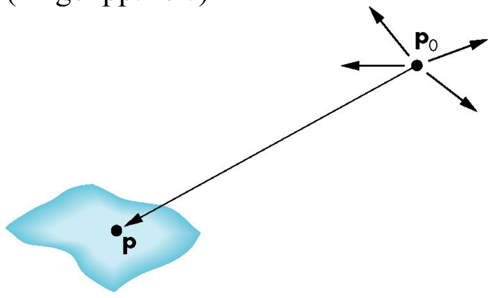
6.2.2 Ambient Light

a. Explain the purpose of the Ambient Light.
ANSWER:
(Angel pp. 295)
6.2.2 Ambient Light In many situations, such as in classrooms or kitchens, the lights have been designed and positioned to provide uniformillumination throughout the room. Often such il-lumination is achieved through large sources that have diffusers whose purpose is to scatter light in all directions. We could create an accurate simulation of such illumina-tion, at least in principle, by modeling all the distributed sources and then integrating the illumination from these sources at each point on a reflecting surface. Making such amodel and rendering a scene with it would be a daunting task for a graphics system, especially one for which real- time performance is desirable. 2 Alternatively, we can look at the desired effect of the sources: to achieve a uniform light level in the room. This uniform lighting is called ambient light. If we follow this second approach, we can postulate an ambient intensity at each point in the environment. Thus, ambient illumination is characterized by an intensity, Ia, that is identical at every point in the scene. Our ambient source has red, green, and blue color components, Lar , Lag , and Lab. We will use the scalar La to denote any one of the three components. Although every point in our scene receives the same illumination from La, each surface can reflect this light differently based on the surfaces properties. Ambient light depends on the color of the the light sources in the environment. For example, a red light bulb in a white room creates red ambient light. Hence, if we turn off the light, the ambient contribution

disappears. OpenGL permits us to add a global ambient term, which does not depend on any of the light sources and is reflected from surfaces. The advantage of adding such a term is that there will always be some light in the environment so that objects in the viewing volume that are not blocked by other objects will always appear in the image.

6.2.3 Point Sources

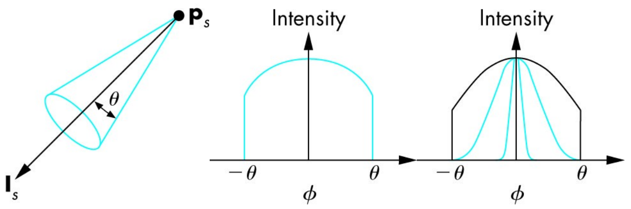
a. Explain Figure 6.7 for point sources.
ANSWER:
(Angel pp. 296)



6.2.3 Point Sources An ideal point source emits light equally in all directions. We can characterize a point source located at a point p0 by a three- component color function L(p0) = (Lr(p0), Lg(p0), Lb(p0)). We use L(p0) to refer to any of the components. The intensity of illumination received from a point source located at p0 at a point p is proportional to the inverse square of the distance from the source. Hence, at a point p (Figure 6.7), any component of the intensity of light received from the point source is given by function of the form L(p, p0) = 1 | p - p0| 2 L(p0). The use of point sources in most applications is determined more by their ease of use than by their resemblance to physical reality. Scenes rendered with only point sources tend to have high contrast; objects appear either bright or dark. In the real world, it is the large size of most light sources that contributes to softer scenes, as we can see from Figure 6.8, which shows the shadows created by a source of finite size. Some areas are fully in shadow, or in the umbra, whereas others are in partial shadow, or in the penumbra. We can mitigate the high- contrast effect from point-source illumination by adding ambient light to a scene. The distance term also contributes to the harsh renderings with point sources. Although the inverse- square distance term is correct for point sources, in practice it is usually replaced by a term of the form (a + bd + cd2)- 1, where d is the distance between p and p0. The constants a, b, and c can be chosen to soften the lighting. In addition, a small amount of ambient light also softens the effect of point sources. Note that if the light source is far from the surfaces in the scene, then the intensity of the light from the source is sufficiently uniform that the distance term is almost constant over the surfaces.

6.2.4 Spotlights

a. Explain Figures 6.9, 6.10, and 6.11 for spotlights.
ANSWER:
(Angel pp. 297)



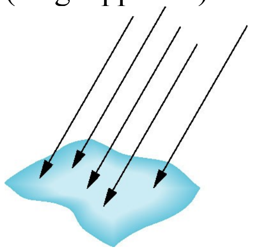
6.2.4 Spotlights Spotlights are characterized by a narrow range of angles through which light is emit-ted. We can construct a simple spotlight from a point source by limiting the angles at which light from the source can be seen. We can use a cone whose apex is at ps, which points in the direction ls, and whose width is determined by an angle ., as shown in Figure 6.9. If . = 180, the spotlight becomes a point source. More realistic spotlights are characterized by the distribution of light within the cone usually with most of the light concentrated in the center of the cone. Thus, the intensity is a function of the angle f between the direction of the source and a vector s to a point on the surface (as long as this angle is less than .; Figure 6.10). Although this function could be defined in many ways, it is usually defined by cose f, where the exponent e (Figure 6.11) determines how rapidly the light intensity drops off. As we will see throughout this chapter, cosines are convenient functions for lighting calculations. If u and v are any unit- length vectors, we can compute the cosine of the angle . between them with the dot product cos . = u . v, a calculation that requires only three multiplications and two additions.

6.2.5 Distant Light Sources

a. Explain Figures 6.12 for parallel light sources.

ANSWER:

(Angel pp. 297)



Most shading calculations require the direction from the point on the surface to the light source position. As we move across a surface, calculating the intensity at each point, we should recompute this vector repeatedly a computation that is a significant part of the lighting calculation. However, if the light source is far from the surface, the vector does not change much as we move from point to point, just as the light from the sun strikes all objects that are in close proximity to one another at the same angle. Figure 6.12 illustrates that we are effectively replacing a point source of light with a source that illuminates objects with parallel rays of light a parallel source. In practice, the calculations for distant light sources are similar to the calculations for parallel projections; they replace the location of the light source with the direction of the light source. Hence, in homogeneous coordinates, the location of a point light source at p0 is represented internally as a four-dimensional column matrix: p0 = ... x y z

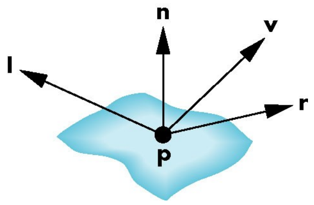
In contrast, the distant light source is described by a direction vector whose represen-tation in homogeneous coordinates is the matrix p0 =..... x y z 0

6.3 THE PHONG LIGHTING MODEL

a. Explain Figures 6.13 for the vectors used by the Phong Model.

ANSWER:

(Angel pp. 298)



The lighting model that we present was introduced by Phong and later modified by Blinn. It has proved to be efficient and to be a close enough approximation to physical reality to produce good renderings under a variety of lighting conditions and material properties. The Phong- Blinn or (modified Phong) model is the basis for lighting and shading in graphics APIs and is implemented on virtually all graphics cards. The Phong lighting model uses the four vectors shown in Figure 6.13 to calculate a color for an arbitrary point p on a surface. If the surface is curved, all four vectors can change as we move from point to point. The vector n is the normal at p; we discuss its calculation in Section 6.4. The vector v is in the direction from p to the viewer or COP. The vector l is in the direction of a line from p to an arbitrary point on the source for a distributed light source or, as we are assuming for now, to the point- light source. Finally, the vector r is in the direction that a perfectly reflected ray from l would take. Note that r is determined by n and l. We calculate it in Section 6.4. The Phong model supports the three types of light material interactions ambient, diffuse, and specular that we introduced in Section 6.1. Suppose that we have a set of point sources. Each source can have separate ambient, diffuse, and spec-ular components for each of the three primary colors. From a physical perspective it may appear somewhat strange that a light source has three independent colors for each of the three types of lighting. However, remem-ber that our goal is to create realistic shading effects in as close to real time as possible, rather than trying to model the BRDF as accurately as possible. Hence, we use tricks with a local model to simulate effects that can be global in nature. The ambient source color represents the the interaction of a light source with the surfaces in the environment whereas the the specular source color is designed to produce the desired color of a specular highlight.
Thus, if our light- source model has separate ambient, diffuse, and specular terms, we need nine coefficients to characterize a light source at any point p on the surface. We can place these nine coefficients in a 3 3 illumination array for the ith light source:
Li =.. Lira Liga Liba Lird Ligd Libd Lirs Ligs Libs .. . The first row of the matrix contains the ambient intensities for the red, green, and blue terms from source i. The second row contains the diffuse terms; the third con-tains the specular terms. We assume that any distance- attenuation terms have not yet been applied. We build our lighting model by summing the contributions for all the light sources at each point we wish to light. For each light source, we have to compute the amount of light reflected for each of the nine terms in the illumination array. For example, for the red diffuse term from source i, Lird, we can compute a reflection term Rird, and the latters contribution to the intensity at p is RirdLird. The value of Rird depends on the material properties, the orientation of the surface, the direction of the light source, and the distance between the light source

and the viewer. Thus, for each point, we have nine coefficients that we can place in an array of reflection terms: Ri =.. Rira Riga Riba Rird Rigd Ribd Rirs Rigs Ribs .. .

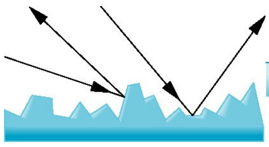
We can then compute the contribution for each color source by adding the ambient, diffuse, and specular components. For example, the red intensity that we see at p from source i is the sum of red ambient, red diffuse, and red specular intensities from this source: Iir = RiraLira + RirdLird + RirsLirs = Iira + Iird + Iirs. We obtain the total intensity by adding the contributions of all sources and, possibly, a global ambient term. Thus, the red term is Ir = i (Iira + Iird + Iirs) + Iar , where Iar is the red component of the global ambient light. We can simplify our notation by noting that the necessary computations are the same for each source and for each primary color. They differ depending on whether we are considering the ambient, diffuse, or specular terms. Hence, we can omit the subscripts i, r, g, and b. We write I = Ia + Id + Is = LaRa + LdRd + LsRs , with the understanding that the computation will be done for each of the primaries and each source; a global ambient term can be added at the end.

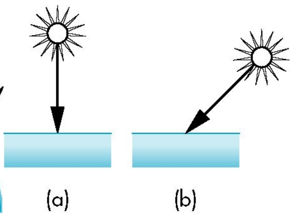
6.3.1 Ambient Reflection

a. Explain Ambient Reflection.
ANSWER:
(Angel pp. 300)
The intensity of ambient light Ia is the same at every point on the surface. Some of this light is absorbed and some is reflected. The amount reflected is given by the ambient reflection coefficient, Ra = ka. Because only a positive fraction of the light is reflected, we must have 0 = ka = 1, and thus Ia = kaLa. Here La can be any of the individual light sources, or it can be a global ambient term. A surface has, of course, three ambient coefficients kar, kag, and kab and they can differ. Hence, for example, a sphere appears yellow under white ambient light

6.3.2 Diffuse Reflection

a. Explain Figures 6.14 and 6.15.
ANSWER:
(Angel pp. 300)





A perfectly diffuse reflector scatters the light that it reflects equally in all directions. Hence, such a surface appears the same to all viewers. However, the amount of light reflected depends both on the material because some of the incoming light is absorbed and on the position of the light source relative to the surface. Diffuse reflections are characterized by rough surfaces. If we were to magnify a cross section of a diffuse surface, we might see an image like that shown in Figure 6.14. Rays of light that hit the surface at only slightly different angles are reflected back at markedly different angles. Perfectly diffuse surfaces are so rough that there is no preferred angle of reflection. Such surfaces, sometimes called Lambertian surfaces, can be modeled mathematically with

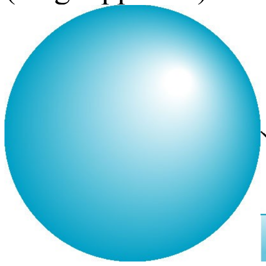
Lamberts law. Consider a diffuse planar surface, as shown in Figure 6.15, illuminated by the sun. The surface is brightest at noon and dimmest at dawn and dusk because, ac- cording to Lamberts law, we see only the vertical component of the incoming light. One way to understand this law is to consider a small parallel light source striking a plane, as shown in Figure 6.16. As the source is lowered in the (artificial) sky, the same amount of light is spread over a larger area, and the surface appears dimmer. Returning to the point source of Figure 6.15, we can characterize diffuse reflections mathematically. Lamberts law states that Rd . cos . , where . is the angle between the normal at the point of interest n and the direction of the light source l. If both l and n are unit- length vectors, 3 then cos . = l . n. If we add in a reflection coefficient kd representing the fraction of incoming diffuse light that is reflected, we have the diffuse reflection term: Id = kd(l . n) Ld. If we wish to incorporate a distance term, to account for attenuation as the light travels a distance d from the source to the surface, we can again use the quadratic attenuation term: Id = kd a + bd + cd2 (l . n) Ld. There is a potential problem with this expression because (l . n) Ld will be neg-ative if the light source is below the horizon. In this case, we want to use zero rather than a negative value. Hence, in practice we use max((l . n) Ld, 0).

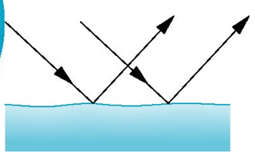
6.3.3 Specular Reflection

a. Explain Figures 6.17 and 6.18.

ANSWER:

(Angel pp. 301)

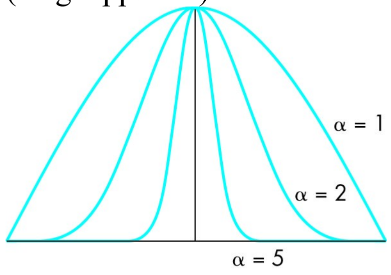




If we employ only ambient and diffuse reflections, our images will be shaded and will appear three- dimensional, but all the surfaces will look dull, somewhat like chalk. What we are missing are the highlights that we see reflected from shiny objects. These highlights usually show a color different from the color of the reflected ambient and diffuse light. For example, a red plastic ball viewed under white light has a white highlight that is the reflection of some of the light from the source in the direction of the viewer (Figure 6.17). Whereas a diffuse surface is rough, a specular surface is smooth. The smoother the surface is, the more it resembles a mirror. Figure 6.18 shows that as the surface gets smoother, the reflected light is concentrated in a smaller range of angles centered FIGURE 6.18 Specular surface.
about the angle of a perfect reflector a mirror or a perfectly specular surface. Mod-eling specular surfaces realistically can be complex because the pattern by which the light is scattered is not symmetric. It depends on the wavelength of the incident light, and it changes with the reflection angle. Phong proposed an approximate model that can be computed with only a slight increase over the work done for diffuse surfaces. The model adds a term for specular reflection. Hence, we consider the surface as being rough for the diffuse term and smooth for the specular term. The amount of light that the viewer sees depends on the angle f between r, the direction of a perfect reflector, and v, the direction of the viewer. The Phong model uses the equation Is = ksLs cosa f. The coefficient ks (0

= ks = 1) is the fraction of the incoming specular light that is reflected.

b. Explain Figures 6.19.
ANSWER:
(Angel pp. 302)



The exponent a is a shininess coefficient. Figure 6.19 shows how, as a is increased, the reflected light is concentrated in a narrower region centered on the angle of a perfect reflector. In the limit, as a goes to infinity, we get a mirror; values in the range 100 to 500 correspond to most metallic surfaces, and smaller values (< 100) correspond to materials that show broad highlights.
The computational advantage of the Phong model is that if we have normalized r and n to unit length, we can again use the dot product, and the specular termbecomes Is = ksLsmax((r . v) a, 0). We can add a distance term, as we did with diffuse reflections. What is referred to as the Phong model, including the distance term, is written I = 1 a + bd + cd2 (kdLdmax(l . n, 0) + ksLsmax((r . v) a, 0)) + kaLa. This formula is computed for each light source and for each primary. It might seem counter intuitive to have a single light source characterized by different amounts of red, green, and blue light for the ambient, diffuse and specular terms in the lighting model. However, the white highlight we might see on a red ball is the distorted reflection of a light source or perhaps some other bright white object in the environment. To calculate this highlight correctly would require a global rather than local lighting calculation. Because we cannot solve the full rendering equation, we instead use various tricks in an attempt to obtain realistic renderings, one of which is to allow different ambient, specular, and diffuse lighting colors. Consider, for example, an environment with many objects. When we turn on a light, some of that light hits a surface directly. These contributions to the image can be modeled with specular and diffuse components of the source. However, much of the rest of the light from the source is scattered from multiple reflections from other objects and makes a contribution to the light received at the surface under considera-tion. We can approximate this termby having an ambient component associated with the source. The shade that we should assign to this term depends on both the color of the source and the color of the objects in the room an unfortunate consequence of our use of approximate models. To some extent, the same analysis holds for diffuse light. Diffuse light reflects among the surfaces, and the color that we see on a partic-ular surface depends on other surfaces in the environment. Again, by using carefully chosen diffuse and specular components with our light sources, we can approximate a global effect with local calculations.
We have developed the Phong lighting model in object space. The actual lighting calculation, however, can be done in a variety of ways within the pipeline. In OpenGL, the default is to, do the calculations for each vertex in eye coordinates and then

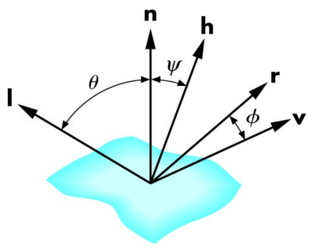
interpolate the shades for each fragment later in the pipeline. We must be careful of the effect of the model- view and projection transformations on the vectors used in the model because these transformations can affect the cosine terms in the model (see Exercise 6.20). Consequently, to make a correct shading calculation, we must either preserve spatial relationships as vertices and vectors pass through the pipeline, perhaps by sending additional information through the pipeline from object space, or go backward through the pipeline to obtain the required shading information.

6.3.4 Modified Phong Model

a. Explain Figures 6.20.

ANSWER:

(Angel pp. 303)



If we use the Phong model with specular reflections in our rendering, the dot product r . v should be recalculated at every point on the surface. We can obtain a different approximation for the specular term by using the unit vector halfway between the view vector and the light- source vector, the halfway vector h = l + v | l + v| . Note that if the normal is in the direction of the halfway vector, then the maximum reflection from the surface is in the direction of the viewer. Figure 6.20 shows all five vectors. Here we have defined . as the angle between n and h, the halfway angle. When v lies in the same plane as do l, n, and r, we can show (see Exercise 6.7) that 2. = f.

If we replace r . v with n . h, we avoid calculation of r. However, the halfway angle . is smaller than f, and if we use the same exponent e in (n . h) e that we used in (r . v) e, then the size of the specular highlights will be smaller. We can mitigate this problem by replacing the value of the exponent e with a value e so that (n . h) e is closer to (r . v) e. It is clear that avoiding recalculation of r is desirable. However, to appreciate fully where savings can be made, you should consider all the cases of flat and curved surfaces, near and far light sources, and near and far viewers (see Exercise 6.8). When we use the halfway vector in the calculation of the specular term, we are using the Blinn- Phong, or modifiedPhong, shading model. This model is the default in OpenGL and is the one carried out on each vertex as it passes down the pipeline. It is important to keep in mind that both the Phong and Blinn- Phong models were created as computationally feasible approximations to the BRDF rather than as the best physical models. The availability of the programmable pipelines that we study in Chapter 9 has opened the door to better approximations of the BRDF that can be implemented on present hardware. Color Plate 25 shows a group of Utah teapots (Section 12.10) that have been ren-dered in OpenGL using the modified Phong model. Note that it is only our ability to control material properties that makes the teapots appear different from one another. The various teapots demonstrate how the modified Phong model can create a variety of surface effects, ranging from dull surfaces to highly reflective surfaces that look like metal.

6.7 LIGHT SOURCES IN OPENGL (see PART B)

6.8 SPECIFICATION OF MATERIALS IN OPENGL(see PART B)

B. (50 pts) Visual Studio 2008 C++ Project

B1. Create Visual Studio 2008 C++, Empty Project, Homework5:
Start with **trackball.c** that we created in Homework 4, also listed below, and create **ShadedCube.c** according to these instructions (name the window `"Shaded Cube"`):

```
/* trackball.c */
/* Rotating cube demo with trackball*/

#include <math.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#define bool int
#define false 0
#define true 1

#ifndef M_PI
#define M_PI 3.14159
#endif


int     winWidth, winHeight;

float   angle = 0.0, axis[3], trans[3];
bool    trackingMouse = false;
bool    redrawContinue = false;
bool    trackballMove = false;

/*----------------------------------------------------------------------*/
/*
** Draw the cube.
*/
GLfloat vertices[][3] = {
   {-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0},
   {-1.0,-1.0,1.0}, {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}
};

GLfloat colors[][3] = {
   {0.0,0.0,0.0},{1.0,0.0,0.0}, {1.0,1.0,0.0}, {0.0,1.0,0.0},
   {0.0,0.0,1.0}, {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}
```

```
};


void polygon(int a, int b, int c , int d, int face)
{

   /* draw a polygon via list of vertices */

   glBegin(GL_POLYGON);
         glColor3fv(colors[a]);
         glVertex3fv(vertices[a]);
         glColor3fv(colors[b]);
         glVertex3fv(vertices[b]);
         glColor3fv(colors[c]);
         glVertex3fv(vertices[c]);
         glColor3fv(colors[d]);
         glVertex3fv(vertices[d]);
   glEnd();
}

void colorcube(void)
{

   /* map vertices to faces */

   polygon(1,0,3,2,0);
   polygon(3,7,6,2,1);
   polygon(7,3,0,4,2);
   polygon(2,6,5,1,3);
   polygon(4,5,6,7,4);
   polygon(5,4,0,1,5);
}

/*----------------------------------------------------------------------*/
/*
** These functions implement a simple trackball-like motion control.
*/

float lastPos[3] = {0.0F, 0.0F, 0.0F};
int curx, cury;
int startX, startY;

void
trackball_ptov(int x, int y, int width, int height, float v[3])
{
   float d, a;
```

```
    /* project x,y onto a hemi-sphere centered within width, height */
    v[0] = (2.0F*x - width) / width;
    v[1] = (height - 2.0F*y) / height;
    d = (float) sqrt(v[0]*v[0] + v[1]*v[1]);
    v[2] = (float) cos((M_PI/2.0F) * ((d < 1.0F) ? d : 1.0F));
    a = 1.0F / (float) sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
    v[0] *= a;
    v[1] *= a;
    v[2] *= a;
}


void
mouseMotion(int x, int y)
{
    float curPos[3], dx, dy, dz;

    trackball_ptov(x, y, winWidth, winHeight, curPos);
if(trackingMouse)
{
    dx = curPos[0] - lastPos[0];
    dy = curPos[1] - lastPos[1];
    dz = curPos[2] - lastPos[2];

    if (dx || dy || dz) {
            angle = 90.0F * sqrt(dx*dx + dy*dy + dz*dz);

            axis[0] = lastPos[1]*curPos[2] - lastPos[2]*curPos[1];
            axis[1] = lastPos[2]*curPos[0] - lastPos[0]*curPos[2];
            axis[2] = lastPos[0]*curPos[1] - lastPos[1]*curPos[0];

            lastPos[0] = curPos[0];
            lastPos[1] = curPos[1];
            lastPos[2] = curPos[2];
    }
}
    glutPostRedisplay();
}

void
startMotion(int x, int y)
{

    trackingMouse = true;
    redrawContinue = false;
```

```
   startX = x; startY = y;
   curx = x; cury = y;
   trackball_ptov(x, y, winWidth, winHeight, lastPos);
         trackballMove=true;
}

void
stopMotion(int x, int y)
{

   trackingMouse = false;

   if (startX != x || startY != y) {
         redrawContinue = true;
   } else {
         angle = 0.0F;
         redrawContinue = false;
         trackballMove = false;
   }
}

/*----------------------------------------------------------------------*/

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

   /* view transform */

   if (trackballMove) {
            glRotatef(angle, axis[0], axis[1], axis[2]);

   }
         colorcube();

   glutSwapBuffers();
}

/*----------------------------------------------------------------------*/

void mouseButton(int button, int state, int x, int y)
{
         if(button==GLUT_RIGHT_BUTTON) exit(0);
         if(button==GLUT_LEFT_BUTTON) switch(state)
         {
   case GLUT_DOWN:
```

```
            y=winHeight-y;
            startMotion( x,y);
            break;
    case GLUT_UP:
            stopMotion( x,y);
            break;
    }
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    winWidth = w;
    winHeight = h;
}

void spinCube()
{
    if (redrawContinue) glutPostRedisplay();
}



void
main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube with trackball");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouseButton);
    glutMotionFunc(mouseMotion);
            glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
            glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    glMatrixMode(GL_MODELVIEW);
    glutMainLoop();
}
```

1. Add the array of GLfloat normals [][3] = { {..}, …. , {..}}; to the 6 faces of the cube (front normal, top normal, left side normal, right side normal, back side normal, bottom side normal).

2. Add the following information about the light source and the materials that you will be using:

```
GLfloat light0_pos[4] = {0.90, 0.90, 2.25, 0.0};

typedef struct materialStruct {
      GLfloat ambient[4];
      GLfloat diffuse[4];
      GLfloat specular[4];
      GLfloat shininess;
      } materialStruct;

typedef struct lightingStruct {
      GLfloat ambient[4];
      GLfloat diffuse[4];
      GLfloat specular [4];
      } lightingStruct;


materialStruct colorCubeMaterials = {
      {0.2, 0.2, 0.2, 1.0},
      {0.8, 0.8, 0.8, 1.0},
      {0.0, 0.0, 0.0, 1.0},
      1.0
      };

materialStruct brassMaterials = {
      {0.33, 0.22, 0.03, 1.0},
      {0.78, 0.57, 0.11, 1.0},
      {0.99, 0.91, 0.81, 1.0},
      27.8
      };

materialStruct redPlasticMaterials = {
      {0.3, 0.0, 0.0, 1.0},
      {0.6, 0.0, 0.0, 1.0},
      {0.8, 0.6, 0.6, 1.0},
      32.0
      };

materialStruct whiteShinyMaterials = {
      {1.0, 1.0, 1.0, 1.0},
      {1.0, 1.0, 1.0, 1.0},
      {1.0, 1.0, 1.0, 1.0},
      100.0
      };

lightingStruct whiteLighting = {
      {0.0, 0.0, 0.0, 1.0},
      {1.0, 1.0, 1.0, 1.0},
      {1.0, 1.0, 1.0, 1.0}
      };

lightingStruct coloredLighting = {
      {0.2, 0.0, 0.0, 1.0},
      {0.0, 1.0, 0.0, 1.0},
      {0.0, 0.0, 1.0, 1.0}
      };


materialStruct *currentMaterials;
lightingStruct *currentLighting;
```

3. In the function colorcube make a call the
glNormal3fv(normals[…]);
before calling polygon function for that face.

4. Add a keyboard callback function called key that on pressing key:
1. calls glutIdleFunc(NULL);
2. calls glutIdleFunc(spinCube);
3. calls currentMaterials = &redPlasticMaterials;
4. calls currentMaterials = &colorCubeMaterials;
5. calls currentMaterials = &brassMaterials;
6. calls currentLighting = &whiteLighting;
7. calls currentLighting = &coloredLighting;

q. calls exit(0);

then it sets the following properties for the material and light

```
glMaterialfv(GL_FRONT, GL_AMBIENT, currentMaterials->ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, currentMaterials->diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, currentMaterials->specular);
glMaterialf(GL_FRONT, GL_SHININESS, currentMaterials->shininess);

glLightfv(GL_LIGHT0, GL_AMBIENT, currentLighting->ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, currentLighting->diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, currentLighting->specular);
```

then it calls glutPostRedisplay();

5. In main call init() function that does the following:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

currentMaterials = &redPlasticMaterials;

glMaterialfv(GL_FRONT, GL_AMBIENT, currentMaterials->ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, currentMaterials->diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, currentMaterials->specular);
glMaterialf(GL_FRONT, GL_SHININESS, currentMaterials->shininess);

currentLighting = &whiteLighting;

glLightfv(GL_LIGHT0, GL_AMBIENT, currentLighting->ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, currentLighting->diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, currentLighting->specular);
glLightfv(GL_LIGHT0, GL_POSITION, light0_pos);
```

Build and run this Project: Insert a screenshot of your output for EACH of the key options and EXPLAIN what happened.
ANSWER:

```
/* ShadedCube.c created from trackball.c */
/* Shading the rotating cube*/

#include <math.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

GLfloat light0_pos[4] = {0.90, 0.90, 2.25, 0.0};

typedef struct materialStruct {
      GLfloat ambient[4];
      GLfloat diffuse[4];
      GLfloat specular[4];
      GLfloat shininess;
      } materialStruct;

typedef struct lightingStruct {
      GLfloat ambient[4];
      GLfloat diffuse[4];
      GLfloat specular [4];
      } lightingStruct;


materialStruct colorCubeMaterials = {
      {0.2, 0.2, 0.2, 1.0},
      {0.8, 0.8, 0.8, 1.0},
      {0.0, 0.0, 0.0, 1.0},
      1.0
      };

materialStruct brassMaterials = {
      {0.33, 0.22, 0.03, 1.0},
      {0.78, 0.57, 0.11, 1.0},
      {0.99, 0.91, 0.81, 1.0},
      27.8
      };

materialStruct redPlasticMaterials = {
      {0.3, 0.0, 0.0, 1.0},
      {0.6, 0.0, 0.0, 1.0},
      {0.8, 0.6, 0.6, 1.0},
      32.0
      };

materialStruct whiteShinyMaterials = {
      {1.0, 1.0, 1.0, 1.0},
      {1.0, 1.0, 1.0, 1.0},
      {1.0, 1.0, 1.0, 1.0},
      100.0
      };

lightingStruct whiteLighting = {
      {0.0, 0.0, 0.0, 1.0},
      {1.0, 1.0, 1.0, 1.0},
      {1.0, 1.0, 1.0, 1.0}
      };

lightingStruct coloredLighting = {
      {0.2, 0.0, 0.0, 1.0},
      {0.0, 1.0, 0.0, 1.0},
      {0.0, 0.0, 1.0, 1.0}
      };


materialStruct *currentMaterials;
lightingStruct *currentLighting;

#define bool int
#define false 0
#define true 1

#ifndef M_PI
#define M_PI 3.14159
#endif


int   winWidth, winHeight;

float       angle = 0.0, axis[3], trans[3];
bool  trackingMouse = false;
bool  redrawContinue = false;
bool    trackballMove = false;

/*----------------------------------------------------------------------
*/
/*
** Draw the cube.
*/
GLfloat vertices[][3] = {
    {-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0},
    {-1.0,-1.0,1.0}, {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}
};

GLfloat colors[][3] = {
    {0.0,0.0,0.0},{1.0,0.0,0.0}, {1.0,1.0,0.0}, {0.0,1.0,0.0},
    {0.0,0.0,1.0}, {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}
};

GLfloat normals[][3] = {{0.0,0.0,-1.0},{0.0,1.0,0.0},
      {-1.0,0.0,0.0}, {1.0,0.0,0.0}, {0.0,0.0,1.0},
      {0.0,-1.0,0.0}};


//void polygon(int a, int b, int c , int d, int face)
//{
//
//    /* draw a polygon via list of vertices */
//
//    glBegin(GL_POLYGON);
//    glColor3fv(colors[a]);
//    glVertex3fv(vertices[a]);
//    glColor3fv(colors[b]);
//    glVertex3fv(vertices[b]);
//    glColor3fv(colors[c]);
//    glVertex3fv(vertices[c]);
//    glColor3fv(colors[d]);
//    glVertex3fv(vertices[d]);
//    glEnd();
//}
//
//void colorcube(void)
//{
//
//    /* map vertices to faces */
//
//    polygon(1,0,3,2,0);
//    polygon(3,7,6,2,1);
//    polygon(7,3,0,4,2);
//    polygon(2,6,5,1,3);
//    polygon(4,5,6,7,4);
//    polygon(5,4,0,1,5);
//}

void polygon(int a, int b, int c , int d)
{

/* draw a polygon via list of vertices */

      glBegin(GL_POLYGON);
            glVertex3fv(vertices[a]);
            glVertex3fv(vertices[b]);
            glVertex3fv(vertices[c]);
            glVertex3fv(vertices[d]);
      glEnd();


                          }

void colorcube(void)
{

/* map vertices to faces */

      glNormal3fv(normals[0]);
      polygon(0,3,2,1);
      glNormal3fv(normals[1]);
      polygon(2,3,7,6);
      glNormal3fv(normals[2]);
      polygon(0,4,7,3);
      glNormal3fv(normals[3]);
      polygon(1,2,6,5);
      glNormal3fv(normals[4]);
      polygon(4,5,6,7);
      glNormal3fv(normals[5]);
      polygon(0,1,5,4);
}
```

```
/*----------------------------------------------------------------------
*/
/*
** These functions implement a simple trackball-like motion control.
*/

float lastPos[3] = {0.0F, 0.0F, 0.0F};
int curx, cury;
int startX, startY;

void
trackball_ptov(int x, int y, int width, int height, float v[3])
{
    float d, a;

    /* project x,y onto a hemi-sphere centered within width, height */
    v[0] = (2.0F*x - width) / width;
    v[1] = (height - 2.0F*y) / height;
    d = (float) sqrt(v[0]*v[0] + v[1]*v[1]);
    v[2] = (float) cos((M_PI/2.0F) * ((d < 1.0F) ? d : 1.0F));
    a = 1.0F / (float) sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
    v[0] *= a;
    v[1] *= a;
    v[2] *= a;
}


void
mouseMotion(int x, int y)
{
    float curPos[3], dx, dy, dz;

    trackball_ptov(x, y, winWidth, winHeight, curPos);
if(trackingMouse)
{
    dx = curPos[0] - lastPos[0];
    dy = curPos[1] - lastPos[1];
    dz = curPos[2] - lastPos[2];

    if (dx || dy || dz) {
      angle = 90.0F * sqrt(dx*dx + dy*dy + dz*dz);

      axis[0] = lastPos[1]*curPos[2] - lastPos[2]*curPos[1];
      axis[1] = lastPos[2]*curPos[0] - lastPos[0]*curPos[2];
      axis[2] = lastPos[0]*curPos[1] - lastPos[1]*curPos[0];

      lastPos[0] = curPos[0];
      lastPos[1] = curPos[1];
      lastPos[2] = curPos[2];
    }
}
    glutPostRedisplay();
}

void
```

```
startMotion(int x, int y)
{

    trackingMouse = true;
    redrawContinue = false;
    startX = x; startY = y;
    curx = x; cury = y;
    trackball_ptov(x, y, winWidth, winHeight, lastPos);
      trackballMove=true;
}

void
stopMotion(int x, int y)
{

    trackingMouse = false;

    if (startX != x || startY != y) {
      redrawContinue = true;
    } else {
      angle = 0.0F;
      redrawContinue = false;
      trackballMove = false;
    }
}

/*----------------------------------------------------------------------
*/

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    /* view transform */

    if (trackballMove) {
          glRotatef(angle, axis[0], axis[1], axis[2]);

    }
      colorcube();

    glutSwapBuffers();
}

/*----------------------------------------------------------------------
*/

void mouseButton(int button, int state, int x, int y)
{
      if(button==GLUT_RIGHT_BUTTON) exit(0);
      if(button==GLUT_LEFT_BUTTON) switch(state)
      {
    case GLUT_DOWN:
      y=winHeight-y;
      startMotion( x,y);
      break;
    case GLUT_UP:
        stopMotion( x,y);
        break;
      }
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    winWidth = w;
    winHeight = h;
}

void spinCube()
{
    if (redrawContinue) glutPostRedisplay();
}

void key(unsigned char k, int x, int y)
{
      switch(k)
      {
            case '1':
                  glutIdleFunc(NULL);
                  break;
            case '2':
                  glutIdleFunc(spinCube);
                  break;
            case '3':
                  currentMaterials = &redPlasticMaterials;
                  break;
            case '4':
                  currentMaterials = &colorCubeMaterials;
                  break;
            case '5':
                  currentMaterials = &brassMaterials;
                  break;
            case '6':
                  currentLighting = &whiteLighting;
                  break;
            case '7':
                  currentLighting = &coloredLighting;
                  break;
            case 'q':
                  exit(0);
                  break;
      }
      glMaterialfv(GL_FRONT, GL_AMBIENT, currentMaterials->ambient);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, currentMaterials->diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, currentMaterials->specular);
      glMaterialf(GL_FRONT, GL_SHININESS, currentMaterials->shininess);

      glLightfv(GL_LIGHT0, GL_AMBIENT, currentLighting->ambient);
      glLightfv(GL_LIGHT0, GL_DIFFUSE, currentLighting->diffuse);
      glLightfv(GL_LIGHT0, GL_SPECULAR, currentLighting->specular);

        glutPostRedisplay();
}
```

```
void init()
{
      glEnable(GL_LIGHTING);
      glEnable(GL_LIGHT0);

      currentMaterials = &redPlasticMaterials;

      glMaterialfv(GL_FRONT, GL_AMBIENT, currentMaterials->ambient);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, currentMaterials->diffuse);
      glMaterialfv(GL_FRONT, GL_SPECULAR, currentMaterials->specular);
      glMaterialf(GL_FRONT, GL_SHININESS, currentMaterials->shininess);

      currentLighting = &whiteLighting;

      glLightfv(GL_LIGHT0, GL_AMBIENT, currentLighting->ambient);
      glLightfv(GL_LIGHT0, GL_DIFFUSE, currentLighting->diffuse);
      glLightfv(GL_LIGHT0, GL_SPECULAR, currentLighting->specular);
      glLightfv(GL_LIGHT0, GL_POSITION, light0_pos);
}


void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Shaded Cube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
      glutKeyboardFunc(key);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouseButton);
    glutMotionFunc(mouseMotion);
      glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
      glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    glMatrixMode(GL_MODELVIEW);
      init();
    glutMainLoop();
}
```